RD
AUDITORS

Catcoin.com Smart Contract, Code Review and Security Analysis Report

Customer: Catcoin.com
Prepared on: 15th February 2023
Platform: Binance
Language: Solidity

rdauditors.com

Table of Contents

Disclaimer

This document may contain confidential information about its systems and intellectual property of the customer as well as information about potential vulnerabilities and methods of their exploitation.

The report containing confidential information can be used internally by the customer or it can be disclosed publicly after all vulnerabilities are fixed - upon the decision of the customer.

Document

Name	Smart Contract Code Review and Security Analysis Report of Catcoin.com
Platform	Binance/ Solidity
File 1	Address.sol
MD5 hash	b89961e443500e3fd00bd2581776dba1
SHA256 hash	d547cab49a97d7f8fd633db312b8a074ef816dd4544af72e4208382e76391647
File 2	Context.sol
MD5 hash	c4b296fb9a98a645ca52cc72c3fbae06
SHA256 hash	6de5302543723d32c8eaf17becc4525936e16d9c4551455c93d306b9b72c0799
File 3	IERC20.sol
MD5 hash	020f718826122fba2f2c83ff2b7cb2cd
SHA256 hash	6654ca211d7ed22937fae539bcf24e0bda89ba7489d4a2f439cc52f53db6ec4d
File 4	iStableStaking.sol
MD5 hash	9874f7a1246794ead37788edfc1c086c

SHA256 hash	2ddaea963694065cede1d31509ac9d0370f047082bfdf11bc8889214582ce53a
File 5	Ownable.sol
MD5 hash	9cc44a70849e3b6acc652e1157d09fbf
SHA256 hash	c53bedd328735571fdf8d130387e2ac3c12a56ea27978e4694b22457b8ab821f
File 6	ReentrancyGuard.sol
MD5 hash	159724b6de9fd97c9b5e28bf38fc12ea
SHA256 hash	d403c9c184c27e1320a5bc543a8efbdc54079110043c827ec513b785c2db20a3
File 7	SafeERC20.sol
MD5 hash	85abf875fb0e10e82d8533c19a0c744a
SHA256 hash	71c37232113f52433042d788efd366ceaecf78d412f009b8a88d776cb6934646
File 8	SafeMath.sol
MD5 hash	d8601ab024d98063d1884414caa798c1
SHA256 hash	8213cd58437a8a6b5acb2a85358cd245f5ae0e44674af84c60a312b8b86049d7
File 9	TokenStaking.sol

MD5 hash	05768299fc9f9815767c23bf960adaea
SHA256 hash	6be87e6d4b124bbd1191183932f9adf5b7c26b2119c86c9c6b69738f25f2d974
Date	15/02/2023

Introduction

RD Auditors (Consultant) were contracted by Catcoin.com (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report represents the findings of the security assessment of the customer`s smart contract and its code review conducted between 13th - 15th February 2023.

This contract consists of nine files.

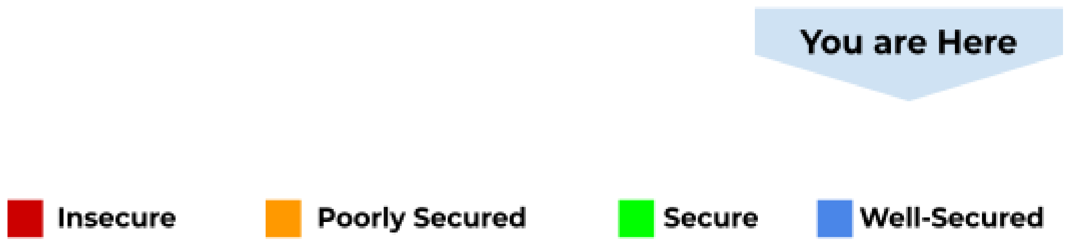
Project Scope

The scope of the project is a smart contract. We have scanned this smart contract for commonly known and more specific vulnerabilities, below are those considered (the full list includes but is not limited to):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Executive Summary

According to the assessment, the customer's solidity smart contract is now **Well-Secured.**



Automated checks are with smartDec, Mythril, Slither and remix IDE. All issues were performed by our team, which included the analysis of code functionality, the manual audit found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the audit overview section. The general overview is presented in the AS-IS section and all issues found are located in the audit overview section.

We found the following;

Total Issues	0
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	0
🟦 Very Low	0

Code Quality

The libraries within this smart contract are part of a logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned to a specific address and its properties/methods can be reused many times by other contracts.

The Catcoin.com team has not provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.

Overall, the code is almost commented. Commenting can provide rich documentation for functions, return variables and more. Use of the Ethereum Natural Language Specification Format (NatSpec) for commenting is recommended.

Documentation

We were given the Catcoin.com code as a link:

https://bscscan.com/address/0x456441ce0cF28aCEd1c326BaBA08814dFD2c37B4#code

The hash of that file is mentioned in the table. As mentioned above, it's recommended to write comments on smart contract code, so anyone can quickly understand the programming flow as well as complex code logic.

Comments are very helpful in understanding the overall architecture of the protocol. It also provides a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure. Those were based on well known industry standard open source projects and even core code blocks that are written well and systematically.

AS-IS Overview

TokenStaking.sol

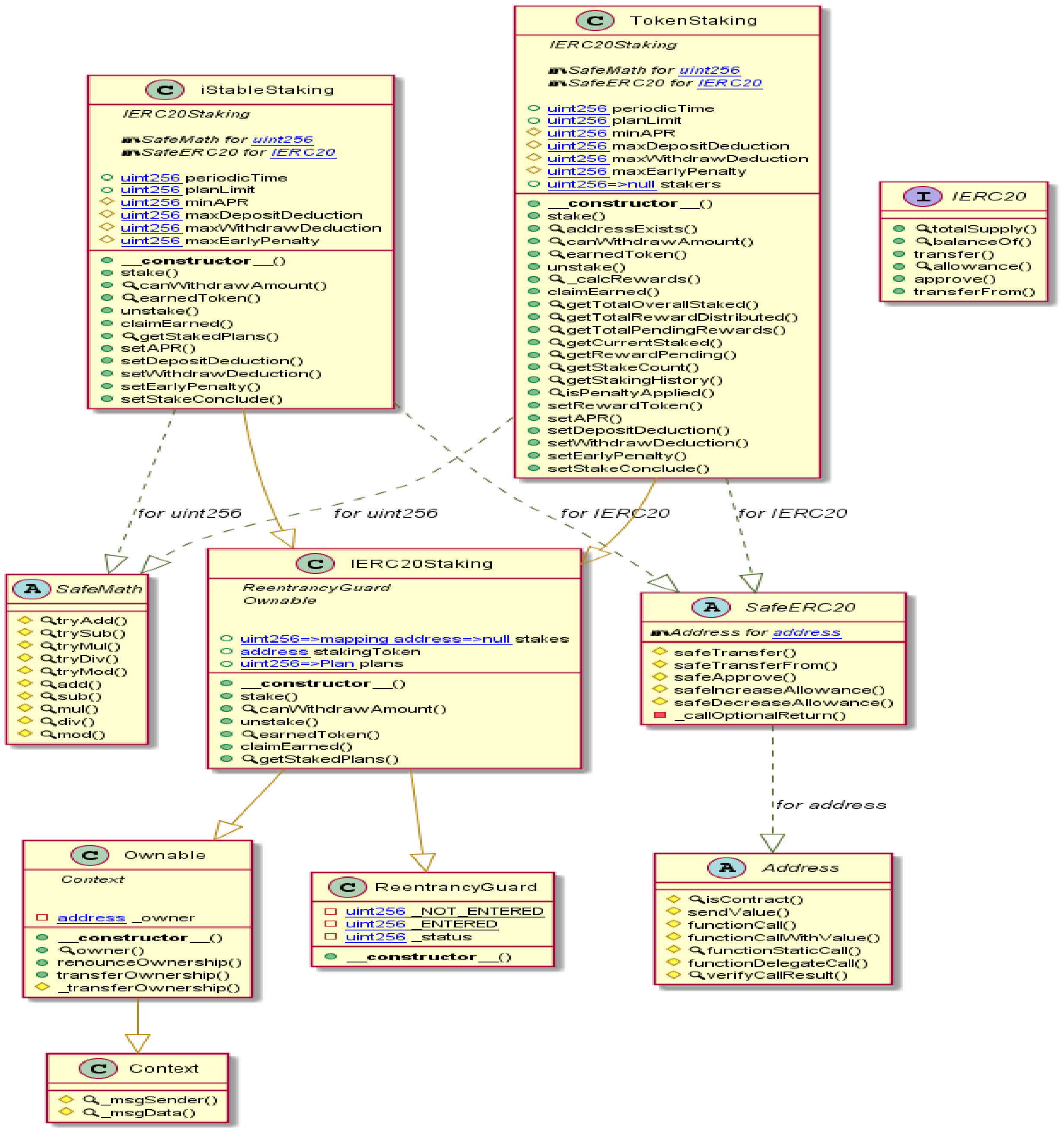
File And Function Level Report

Contract:	TokenStaking
Inherit:	IERC20Staking
Observation:	Passed
Test Report:	Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	`stake`	public	Passed	All Passed	No Issue	Passed
2	`addressExists`	public	Passed	All Passed	No Issue	Passed
3	`canWithdrawAmount`	public	Passed	All Passed	No Issue	Passed
4	`earnedToken`	public	Passed	All Passed	No Issue	Passed
5	`unstake`	public	Passed	All Passed	No Issue	Passed
6	`_calcRewards`	public	Passed	All Passed	No Issue	Passed
7	`claimEarned`	public	Passed	All Passed	No Issue	Passed
8	`getTotalOverallStaked`	public	Passed	All Passed	No Issue	Passed
9	`getTotalRewardDistributed`	public	Passed	All Passed	No Issue	Passed
10	`getTotalPendingRewards`	public	Passed	All Passed	No Issue	Passed
11	`getCurrentStaked`	public	Passed	All Passed	No Issue	Passed
12	`getRewardPending`	public	Passed	All Passed	No Issue	Passed

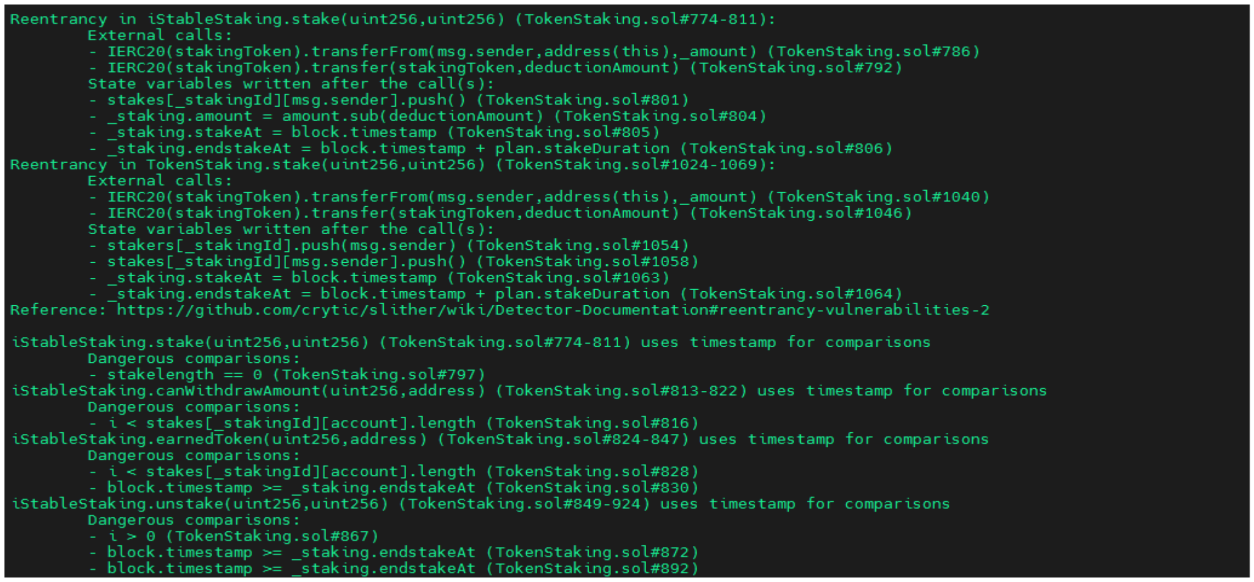
13	`getStakeCount`	public	Passed	All Passed	No Issue	Passed
14	`getStakingHistory`	public	Passed	All Passed	No Issue	Passed
15	`isPenaltyApplied`	public	Passed	All Passed	No Issue	Passed
16	`setRewardToken`	external	Passed	All Passed	No Issue	Passed
17	`setAPR`	external	Passed	All Passed	No Issue	Passed
18	`setDepositDeduction`	external	Passed	All Passed	No Issue	Passed
19	`setWithdrawDeduction`	external	Passed	All Passed	No Issue	Passed
20	`setEarlyPenalty`	external	Passed	All Passed	No Issue	Passed
21	`setStakeConclude`	external	Passed	All Passed	No Issue	Passed

Code Flow Diagram - TokenStaking.sol

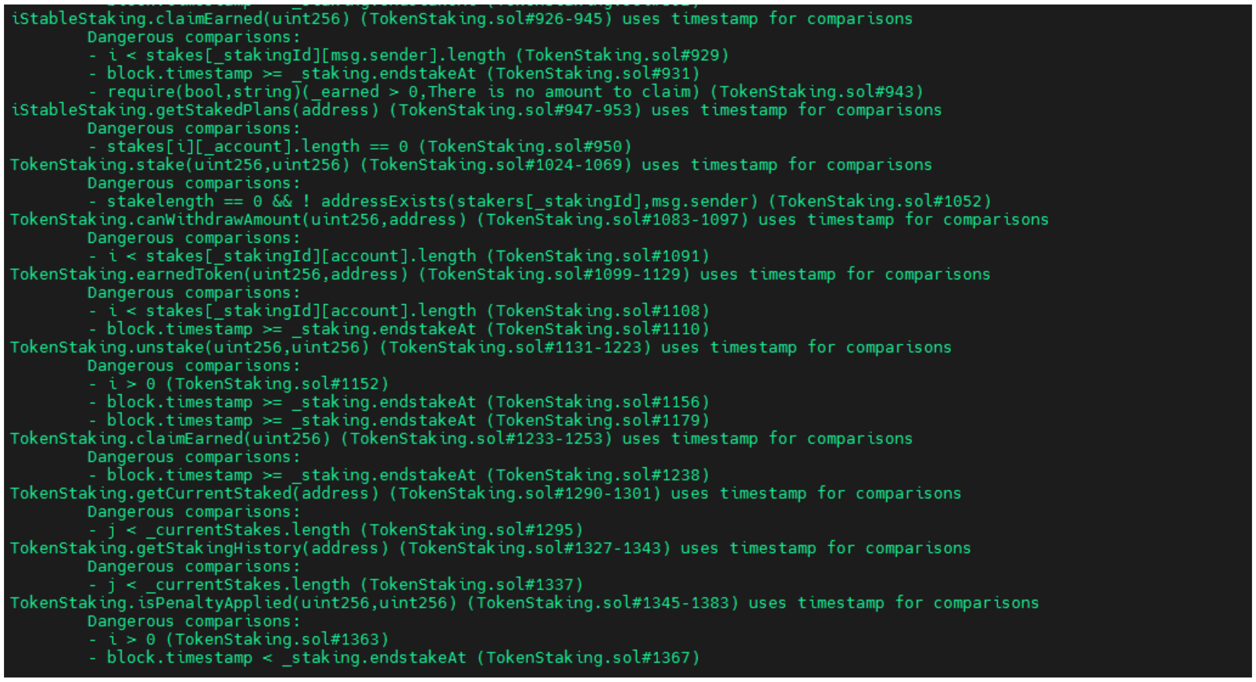


Code Flow Diagram - Slither Results Log

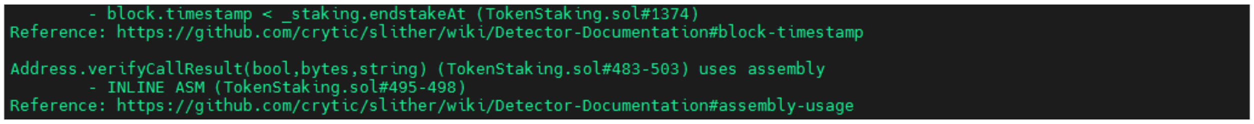
TokenStaking.sol



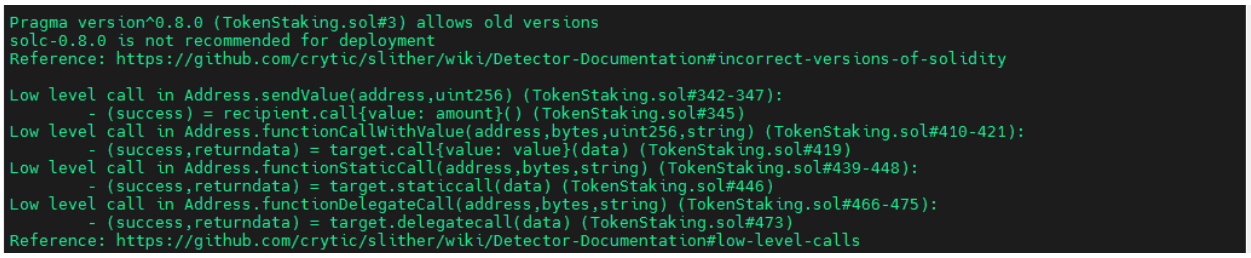
```
Reentrancy in iStableStaking.stake(uint256,uint256) (TokenStaking.sol#774-811):
        External calls:
        - IERC20(stakingToken).transferFrom(msg.sender,address(this),_amount) (TokenStaking.sol#786)
        - IERC20(stakingToken).transfer(stakingToken,deductionAmount) (TokenStaking.sol#792)
        State variables written after the call(s):
        - stakes[_stakingId][msg.sender].push() (TokenStaking.sol#801)
        - _staking.amount = amount.sub(deductionAmount) (TokenStaking.sol#804)
        - _staking.stakeAt = block.timestamp (TokenStaking.sol#805)
        - _staking.endstakeAt = block.timestamp + plan.stakeDuration (TokenStaking.sol#806)
Reentrancy in TokenStaking.stake(uint256,uint256) (TokenStaking.sol#1024-1069):
        External calls:
        - IERC20(stakingToken).transferFrom(msg.sender,address(this),_amount) (TokenStaking.sol#1040)
        - IERC20(stakingToken).transfer(stakingToken,deductionAmount) (TokenStaking.sol#1046)
        State variables written after the call(s):
        - stakers[_stakingId].push(msg.sender) (TokenStaking.sol#1054)
        - stakes[_stakingId][msg.sender].push() (TokenStaking.sol#1058)
        - _staking.stakeAt = block.timestamp (TokenStaking.sol#1063)
        - _staking.endstakeAt = block.timestamp + plan.stakeDuration (TokenStaking.sol#1064)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2

iStableStaking.stake(uint256,uint256) (TokenStaking.sol#774-811) uses timestamp for comparisons
        Dangerous comparisons:
        - stakelength == 0 (TokenStaking.sol#797)
iStableStaking.canWithdrawAmount(uint256,address) (TokenStaking.sol#813-822) uses timestamp for comparisons
        Dangerous comparisons:
        - i < stakes[_stakingId][account].length (TokenStaking.sol#816)
iStableStaking.earnedToken(uint256,address) (TokenStaking.sol#824-847) uses timestamp for comparisons
        Dangerous comparisons:
        - i < stakes[_stakingId][account].length (TokenStaking.sol#828)
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#830)
iStableStaking.unstake(uint256,uint256) (TokenStaking.sol#849-924) uses timestamp for comparisons
        Dangerous comparisons:
        - i > 0 (TokenStaking.sol#867)
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#872)
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#892)
```



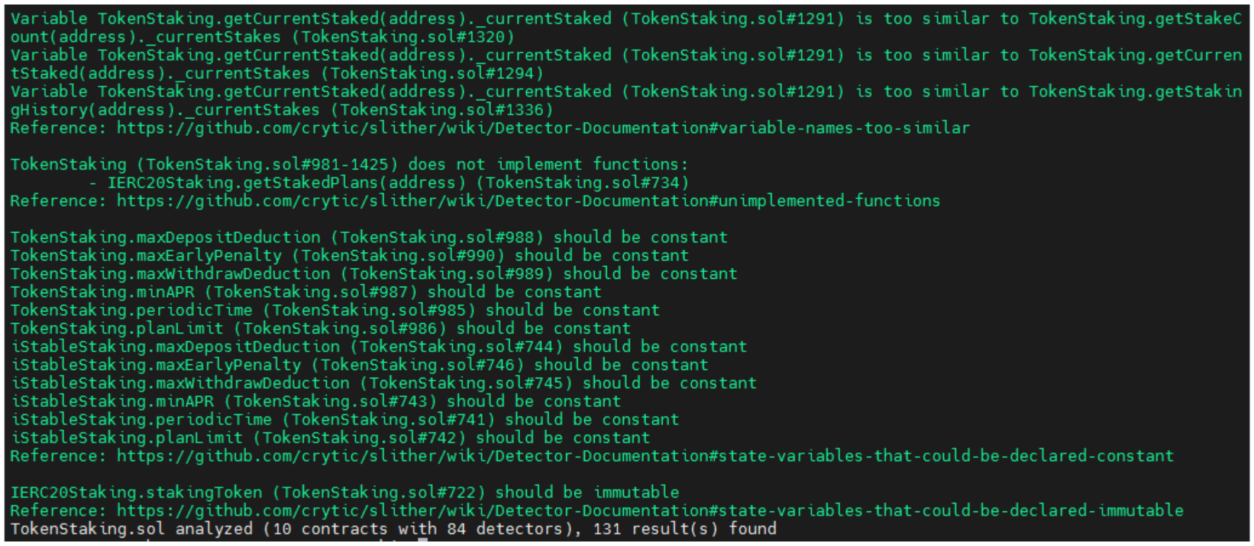
```
iStableStaking.claimEarned(uint256) (TokenStaking.sol#926-945) uses timestamp for comparisons
        Dangerous comparisons:
        - i < stakes[_stakingId][msg.sender].length (TokenStaking.sol#929)
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#931)
        - require(bool,string)(_earned > 0,There is no amount to claim) (TokenStaking.sol#943)
iStableStaking.getStakedPlans(address) (TokenStaking.sol#947-953) uses timestamp for comparisons
        Dangerous comparisons:
        - stakes[i][_account].length == 0 (TokenStaking.sol#950)
TokenStaking.stake(uint256,uint256) (TokenStaking.sol#1024-1069) uses timestamp for comparisons
        Dangerous comparisons:
        - stakelength == 0 && ! addressExists(stakers[_stakingId],msg.sender) (TokenStaking.sol#1052)
TokenStaking.canWithdrawAmount(uint256,address) (TokenStaking.sol#1083-1097) uses timestamp for comparisons
        Dangerous comparisons:
        - i < stakes[_stakingId][account].length (TokenStaking.sol#1091)
TokenStaking.earnedToken(uint256,address) (TokenStaking.sol#1099-1129) uses timestamp for comparisons
        Dangerous comparisons:
        - i < stakes[_stakingId][account].length (TokenStaking.sol#1108)
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#1110)
TokenStaking.unstake(uint256,uint256) (TokenStaking.sol#1131-1223) uses timestamp for comparisons
        Dangerous comparisons:
        - i > 0 (TokenStaking.sol#1152)
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#1156)
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#1179)
TokenStaking.claimEarned(uint256) (TokenStaking.sol#1233-1253) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp >= _staking.endstakeAt (TokenStaking.sol#1238)
TokenStaking.getCurrentStaked(address) (TokenStaking.sol#1290-1301) uses timestamp for comparisons
        Dangerous comparisons:
        - j < _currentStakes.length (TokenStaking.sol#1295)
TokenStaking.getStakingHistory(address) (TokenStaking.sol#1327-1343) uses timestamp for comparisons
        Dangerous comparisons:
        - j < _currentStakes.length (TokenStaking.sol#1337)
TokenStaking.isPenaltyApplied(uint256,uint256) (TokenStaking.sol#1345-1383) uses timestamp for comparisons
        Dangerous comparisons:
        - i > 0 (TokenStaking.sol#1363)
        - block.timestamp < _staking.endstakeAt (TokenStaking.sol#1367)
```



```
        - block.timestamp < _staking.endstakeAt (TokenStaking.sol#1374)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address.verifyCallResult(bool,bytes,string) (TokenStaking.sol#483-503) uses assembly
        - INLINE ASM (TokenStaking.sol#495-498)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
```



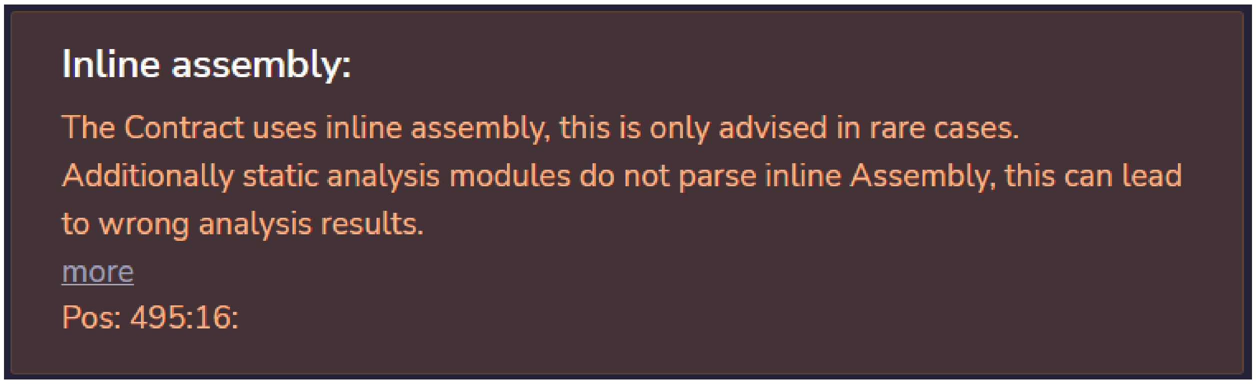
```
Pragma version^0.8.0 (TokenStaking.sol#3) allows old versions
solc-0.8.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in Address.sendValue(address,uint256) (TokenStaking.sol#342-347):
        - (success) = recipient.call{value: amount}() (TokenStaking.sol#345)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (TokenStaking.sol#410-421):
        - (success,returndata) = target.call{value: value}(data) (TokenStaking.sol#419)
Low level call in Address.functionStaticCall(address,bytes,string) (TokenStaking.sol#439-448):
        - (success,returndata) = target.staticcall(data) (TokenStaking.sol#446)
Low level call in Address.functionDelegateCall(address,bytes,string) (TokenStaking.sol#466-475):
        - (success,returndata) = target.delegatecall(data) (TokenStaking.sol#473)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
```



```
Variable TokenStaking.getCurrentStaked(address)._currentStaked (TokenStaking.sol#1291) is too similar to TokenStaking.getStakeC
ount(address)._currentStakes (TokenStaking.sol#1320)
Variable TokenStaking.getCurrentStaked(address)._currentStaked (TokenStaking.sol#1291) is too similar to TokenStaking.getCurren
tStaked(address)._currentStakes (TokenStaking.sol#1294)
Variable TokenStaking.getCurrentStaked(address)._currentStaked (TokenStaking.sol#1291) is too similar to TokenStaking.getStakin
gHistory(address)._currentStakes (TokenStaking.sol#1336)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-too-similar

TokenStaking (TokenStaking.sol#981-1425) does not implement functions:
        - IERC20Staking.getStakedPlans(address) (TokenStaking.sol#734)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unimplemented-functions

TokenStaking.maxDepositDeduction (TokenStaking.sol#988) should be constant
TokenStaking.maxEarlyPenalty (TokenStaking.sol#990) should be constant
TokenStaking.maxWithdrawDeduction (TokenStaking.sol#989) should be constant
TokenStaking.minAPR (TokenStaking.sol#987) should be constant
TokenStaking.periodicTime (TokenStaking.sol#985) should be constant
TokenStaking.planLimit (TokenStaking.sol#986) should be constant
iStableStaking.maxDepositDeduction (TokenStaking.sol#744) should be constant
iStableStaking.maxEarlyPenalty (TokenStaking.sol#746) should be constant
iStableStaking.maxWithdrawDeduction (TokenStaking.sol#745) should be constant
iStableStaking.minAPR (TokenStaking.sol#743) should be constant
iStableStaking.periodicTime (TokenStaking.sol#741) should be constant
iStableStaking.planLimit (TokenStaking.sol#742) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant

IERC20Staking.stakingToken (TokenStaking.sol#722) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
TokenStaking.sol analyzed (10 contracts with 84 detectors), 131 result(s) found
```

Solidity Static Analysis

TokenStaking.sol

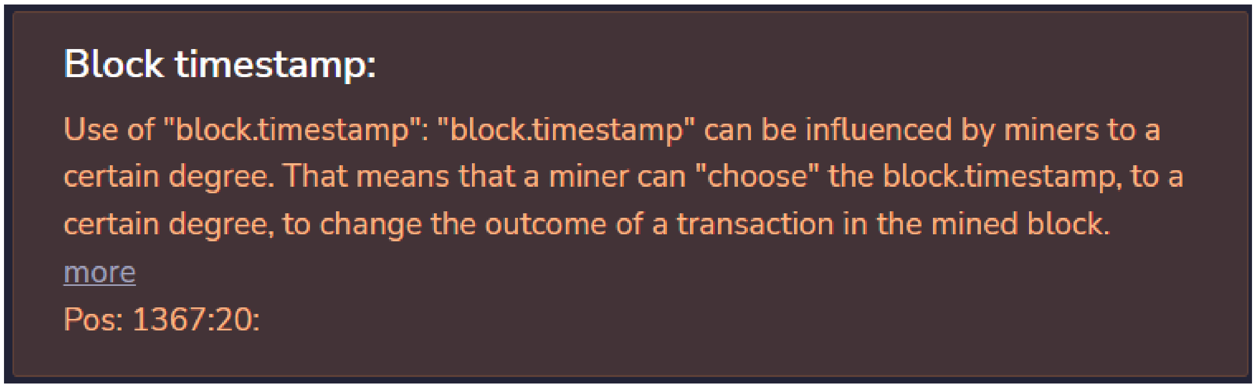


Inline assembly:

The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.

more

Pos: 495:16:

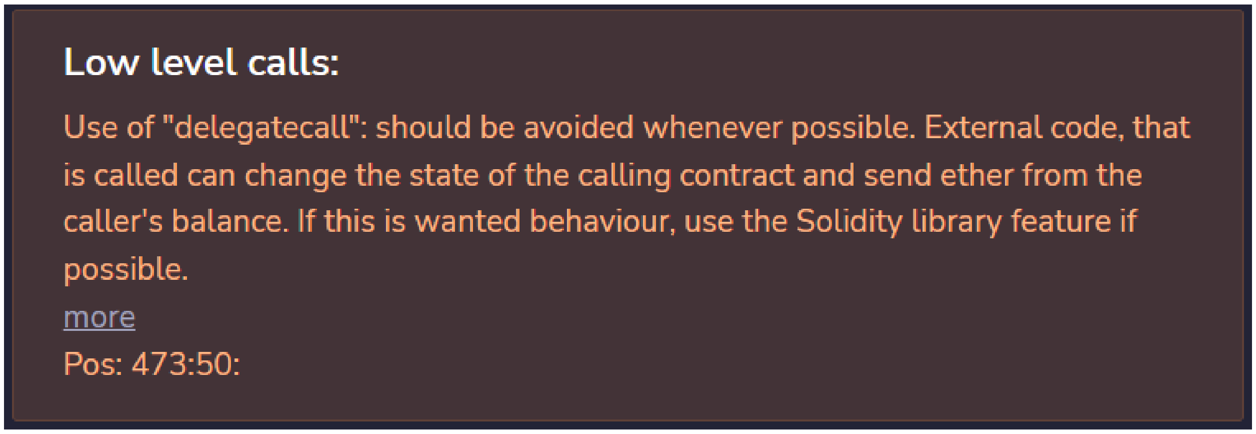


Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 1367:20:

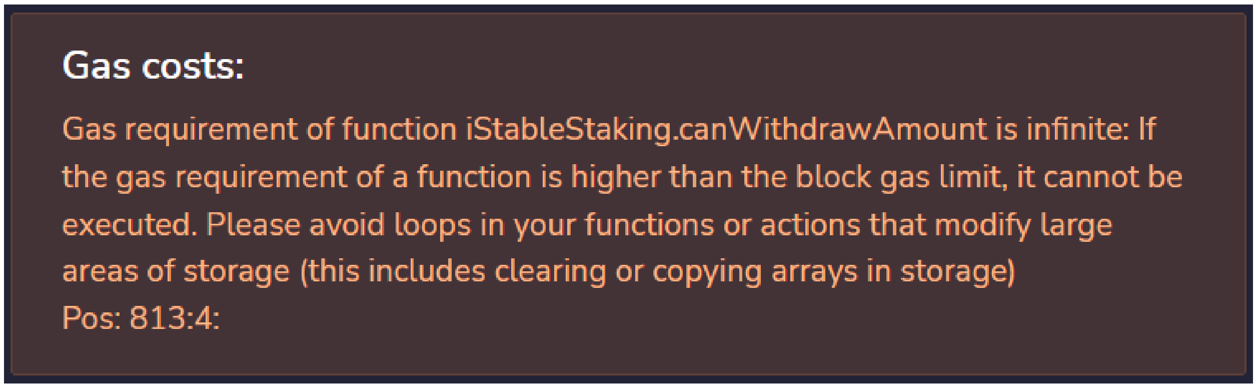


Low level calls:

Use of "delegatecall": should be avoided whenever possible. External code, that is called can change the state of the calling contract and send ether from the caller's balance. If this is wanted behaviour, use the Solidity library feature if possible.

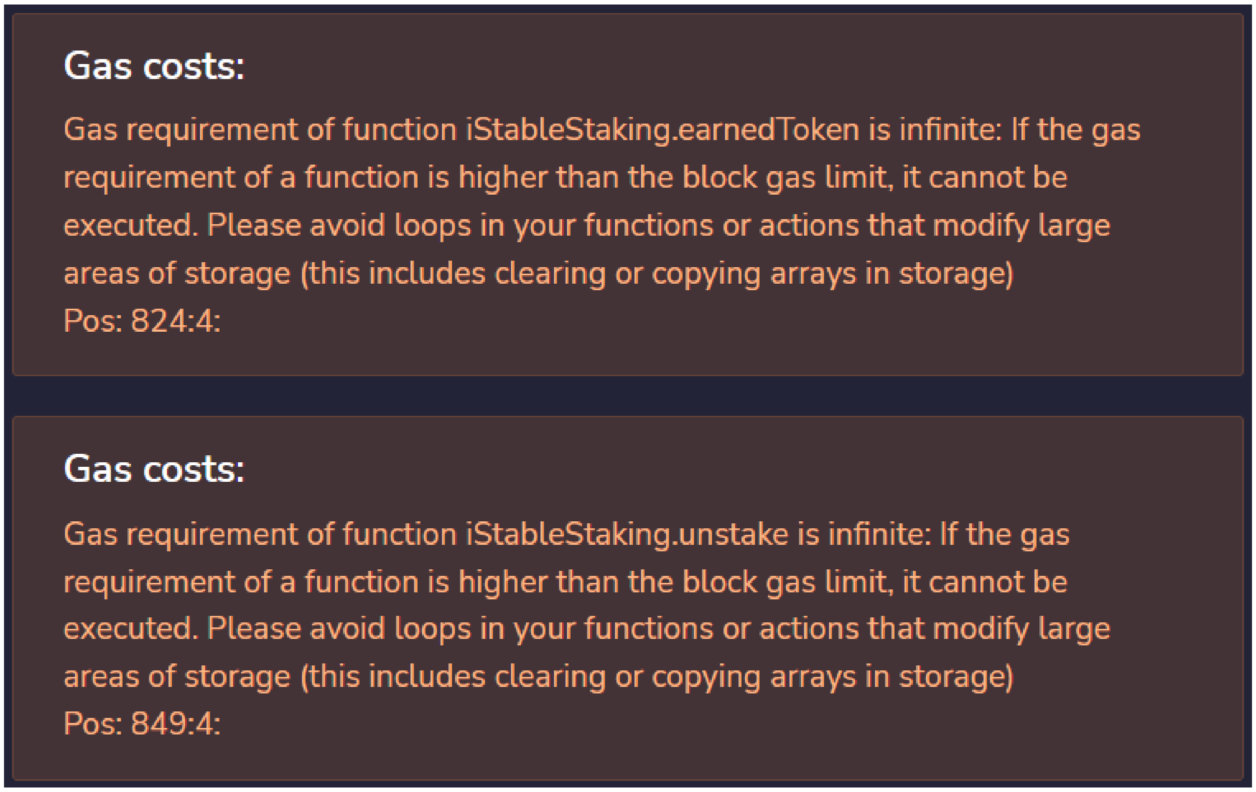
more

Pos: 473:50:



Gas costs:

Gas requirement of function iStableStaking.canWithdrawAmount is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 813:4:



Gas costs:

Gas requirement of function iStableStaking.earnedToken is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 824:4:

Gas costs:

Gas requirement of function iStableStaking.unstake is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 849:4:

Gas costs:

Gas requirement of function iStableStaking.claimEarned is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 926:4:

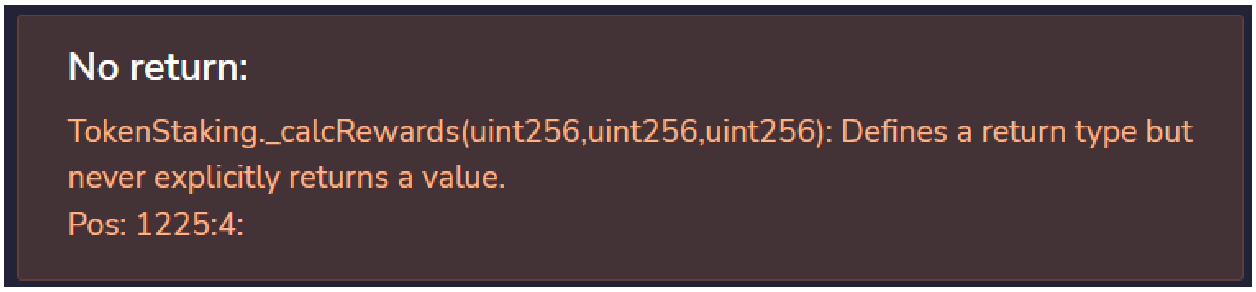
For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
more
Pos: 1091:8:

Miscellaneous

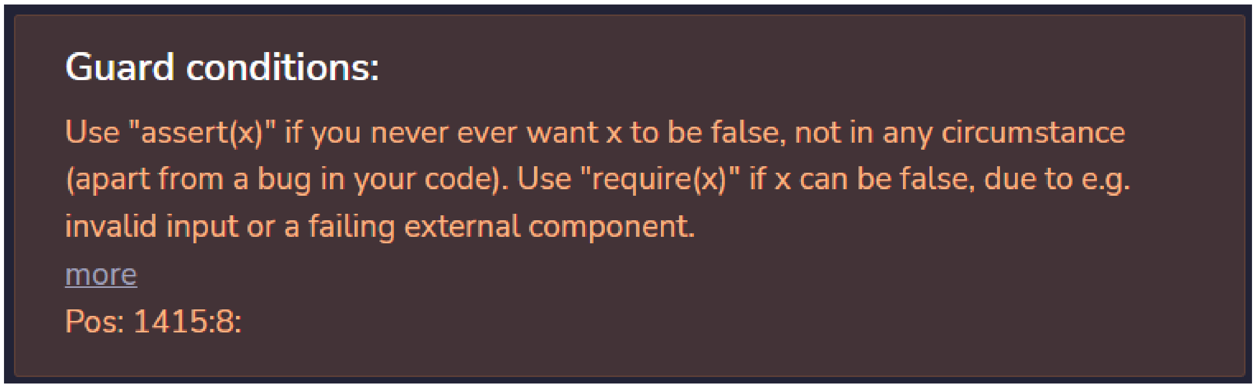
Similar variable names:

TokenStaking.isPenaltyApplied(uint256,uint256) : Variables have very similar names "_staking" and "_stakingId". Note: Modifiers are currently not considered by this static analysis.
Pos: 1374:38:



No return:

TokenStaking._calcRewards(uint256,uint256,uint256): Defines a return type but never explicitly returns a value.
Pos: 1225:4:



Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
more
Pos: 1415:8:

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to lost tokens etc.
High	High level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial functions.
Medium	Medium level vulnerabilities are important to fix; however, they cannot lead to lost tokens.
Low	Low level vulnerabilities are most related to outdated, unused etc. These code snippets cannot have a significant impact on execution.
Lowest Code Style/ Best Practice	Lowest level vulnerabilities, code style violations and information statements cannot affect smart contract execution and can be ignored.

Audit Findings

Critical:

No critical severity vulnerabilities were found.

High:

No high severity vulnerabilities were found.

Medium:

No medium severity vulnerabilities were found.

Low:

No low severity vulnerabilities were found.

Very Low:

No very low severity vulnerabilities were found.

Conclusion

We were given a contract file and have used all possible tests based on the given object. So it is now ready for mainnet deployment. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

The security state of the reviewed contract is “**Well-Secured**”.

Note For Contract Users

There are several owner only functions. Those can be called by the owner's wallet only. So, if the owner's wallet is compromised, then it carries the risk of the contract becoming vulnerable.

Owner has full control over the smart contract. Thus, technical auditing does not guarantee the project's ethical side.

Please do your due diligence before investing. Our audit report is never an investment advice.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyse the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

RD Auditors Disclaimer

The smart contracts given for audit have been analysed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Because the total number of test cases are unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

RD

AUDITORS

Email: info@rdauditors.com

Website: www.rdauditors.com